RT-MINIXv2: Architecture and Interrupt Handling

Pablo A. Pessolani
Facultad Regional Santa Fe
Departamento de Sistemas de Información
Universidad Tecnológica Nacional
Lavaise 610. (3000) Santa Fe. Argentina
ppessolani@hotmail.com

Abstract

Tanenbaum's MINIX operating system [1] was extended by Wainer with Real-Time (RT) services to conform RT-MINIX [2,3]. This work is on RT-MINIXv2, a new version for academic uses that includes a RT-microkernel with more flexible IPC facilities supporting basic priority inheritance protocol, a fixed priority scheduler, timer and event driven interrupt management, statistics and RT-metrics collection keeping backward compatibility with standard MINIX versions.

Keywords: Operating Systems, Minix, Interrupt Handling, Real-Time.

1. Introduction

Computer science students and professionals working on Real-Time Operating Systems (RTOS) need a deep knowledge of every software component and the interactions with hardware devices considering timing constraints.

RTOS instructors can choice among commercial or free licence software to develop their laboratory practice. Commercially available RTOS are too costly and proprietary to be used by academic institutions. Free licence and open source RTOS have been designed focusing on predictability as a key design feature with complex source code readability.

The design and implementation of RT-MINIXv2 is proposed to teach RTOS and covers the following topics:

4 MINIX and RT-MINIX Background Information
4 System Architecture
4 Interrupt Handling
4 Process Management
4 Time Management
4 Process Scheduling
4 Interprocess Communications
4 Real-Time System Calls
4 Statistics Collection
4 Performance Tests
4 Sample Source Code and System Data Structures

This article address the first three topics raised above and describes a way to transform a time sharing OS (MINIX) into a hard Real-Time one (RT-MINIXv2).

The experience earned in well-planed course assignments and projects using systems like RT-MINIXv2 provides education, advanced technical training, and enhances personal performance for the deployment and use of RTOS to the academic community worldwide.

The rest of this work is organized as follows. Section 2 describes the motivation of this project. Section 3 and 4 are about MINIX and RT-MINIX features respectively. Section 5, the proposed RT-MINIXv2 is introduced. Section 6 describes MINIX architecture, and the proposed RT-MINIXv2 architecture is discussed in Section 7. The Section 8 is the longest and covers the RT-MINIXv2 interrupt handling approach, its Interrupt Request Level architecture, the effect of priorities on interrupt service, software IRQs, nested interrupts and a technique to estimate interrupt handlers latency inside the kernel. Finally, Section 9 amd 10 presents conclusions future works respectively.

2. Motivation

The aim of the RT-MINIXv2 project is to provide an educational tool for RTOS courses as MINIX [7, 8, 9] or Linux [6] do for OS Design and Implementation courses.

The decision to adopt MINIX among other OS as foundation for this work is based on:

4 The author's previous experience.

- Its documentation availability.
- Its hardware platform requirements.
- It's modular and elegant design.

RT-MINIXv2 implementation focus on source code readability (perhaps sacrificing performance) to let instructors make easily a multiplicity of grade courses assignments, laboratory tests, projects and other academic uses with an open source RTOS.

Some interesting projects could be:

- Port a hard real-time network protocol stack as RTNET [13] or RETHER [14]
- Design a real-time Distributed Operating System.
- Implement Remote Device Drivers

A lot of interesting system statistics are collected to make the OS more educational.

Students can experience with programming interrupt-driven systems and get a deep understanding of how RT-systems work with a minimal software infrastructure.

3. MINIX Time Sharing Features

MINIX is a complete, time-sharing, multitasking OS developed from scratch by Andrew S. Tanenbaum[1]. It is a general-purpose OS broadly used in Computer Science degree courses.

Though it is copyrighted, the source has been made widely available to universities for study and research. Its main features are:

- *Microkernel based*: Provides process management and scheduling, basic memory management, interprocess communication, interrupt processing and low level I/O support.
- *Multilayer system:* Permits a modular design and clear implementation.
- *Client/Server model*: All system services and device drivers are implemented as server processes with their own execution environment.
- *Message Transfer Interprocess Communications (IPC):* Used for process synchronization and data sharing.
- *Interrupt hiding:* Interrupts are converted in message transfers.

4. RT-MINIX Features

Gabriel Wainer [2, 3] changed the standard MINIX OS to support RT-processing named it "RT-MINIX". Its main features are:

- Scheduling Algorithms Selection
- Joined Scheduling Queues
- Real-Time Metrics collection
- Timer Resolution Management

Several changes was made to MINIX source code of the kernel in order to provide the user a set of system calls to create and manage tasks, both periodic or aperiodic.

That approach implies some limitations in its uses because:

- It is not an architecture, is a patch for MINIX.
- It does not serve hardware interrupts by priority. This could produce priority inversion because while a higher priority RT-handler is running, lower priority NRT-interrupts could be attended increasing the RT-handler interference.
- It has only one level of priority for all RT-tasks, but MINIX tasks and servers have higher priorities. This could produce priority inversion because while a RT-process is running, a standard MINIX non real-time task or server could preempt it.
- Its use standard MINIX message transfers as its IPC primitives. As MINIX use FIFO discipline to receive messages from several processes offen creates a priority inversion problem.
- It does not have any protocol agains priority inversion limiting its utilization on projects that use cooperating tasks.
- Increasing the timer resolution also increase the overhead because the standard MINIX timer interrupt handler is executing at higher frequency.
- A RT-process can make NRT-system calls, therefore the NRT-servers that receive the request will execute in behalf of the RT-process but without RT-attributes.
- As its uses a modified MINIX clock (timer) handler for RT-alarms, other NRT-tasks in ready to run state could be executed before the clock handler increasing the RT-alarms latency.

RM and EDF scheduling were included. These strategies were later combined with other traditional strategies, such as Least Laxity First, Least Slack First and Deadline Monotonic.

Several data structures in the OS were modified to consider tasks period, execution time and criticality.

Task execution priority was implemented using a multiqueue scheme to accommodate RT-tasks along with interactive and CPU-bound tasks.

RT-MINIX defines a new set of signals to indicate special situations, such as missed deadlines, overload or uncertainty of the schedulability of the task set.

5. RT-MINIXv2 Features

As Wainer and Rogina said in [2], existing RTOS can be divided in two categories:

1. Systems implemented using somewhat stripped down and optimized (or specialized) versions of conventional timesharing OS.
2. Systems starting from scratch, focusing on predictability as a key design feature.

RT-MINIXv2 is based on the first design approach using MINIX as the conventional OS.

This special version offers predictable RT-computing environment at lower cost than propietary RTOS used for academic purposes.

The major features of RT-MINIXv2 are summarized as follows:

4 Real-Time preemptive Microkernel.
4 Layered Architecture.
4 Timer/Event Driven Interrupt Management.
4 Fixed Priority Hardware Interrupt Processing.
4 Two Stages Interrupt Handling using Software IRQs.
4 Periodic and Sporadic processing.
4 Timer Resolution Management detached from MINIX timer.
4 Fixed Priority Based Real-Time Scheduling.
4 Synchronous/Asynchronous Message Transfer using Mailboxes.
4 Basic Priority Inheritance Protocol support in Message Transfer IPC.
4 Receive and Synchronous Send Timeout support.
4 Priority discipline on message reception. Deadline expiration watchdogs.
4 Software timers for alarms, timeouts and other time related uses.
4 Statistics and Real-Time Metrics Collection.

The advantage of using a microkernel for RT-applications is that the preemptability of the kernel is better, the size of the kernel becomes much smaller, and the addition/removal of services is easier.

RT-MINIXv2 provides the capability of running special RT-tasks and interrupt handlers on the same machine as standard MINIX. These tasks and handlers execute when they need to execute no matter what MINIX is doing.

The RT-microkernel works by treating the MINIX OS kernel as a task executing under a small RTOS based on software emulation of interrupt control hardware. In fact, MINIX is like the idle task for the RT-microkernel executing only when there are no RT-tasks to run. When MINIX tells the hardware to disable interrupts, the RT-microkernel intercepts the request, records it, and returns to MINIX.

RT-MINIXv2 can handle devices in two ways:

4 *Interrupt driven:* As RT-MINIXv2 is designed for 32 bits INTEL PCs [5], the interrupt levels of the standard devices like keyboard, serial ports, etc cannot be changed and may produce priority inversion problems with other devices.
4 *Timer driven:* the devices are polled or some job is done on the device at regular periods. As in standards PCs the timer has the top priority level, no priority inversion problems can occurs.

The current version of RT-MINIXv2 is based on version 2.0.2 for 32 bits INTEL processors of MINIX therefore it requires the same hardware platform.

6. MINIX System Architecture

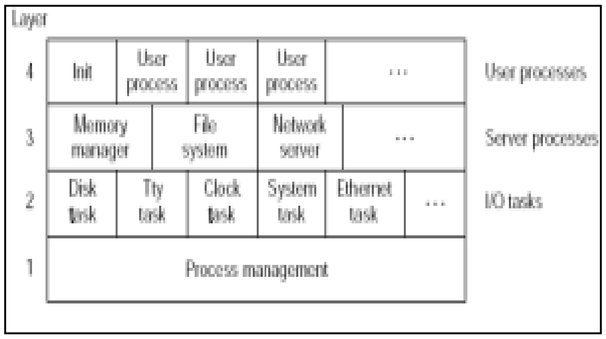

Figure 1- MINIX architecture (from [1])

MINIX is structured in four layers as it can see in Figure 1.

1. The microkernel.
2. Input/Output Tasks.
3. Server Processes.
4. User-level Processes.

An interrupt is usually defined as an event that alters the sequence of instructions executed by a processor [10].

In MINIX, when a hardware device interrupts the CPU, an interrupt handler is called, but if more time is needed to complete the job, the handler sends a message to the device Interrupt Service Task (IST)

and calls the scheduler on exit. As I/O Tasks have greater priority than regular User-level processes and system servers, the IST is executed to resume the interrupt service.
This approach is often called Two Stages Interrupt Handling.

An IST is like a thread that shares kernel address space but it has its own processing attributes.

The use of ISTs to complete the interrupt processing works well in a time sharing environment but can introduce unbounded delay in RT-processing. Two factors affects the interrupt response time:

1. MINIX scheduler uses three priority queues, one for ISTs, one for Server Processes and one for User-level Processes. As the IST queue is arranged in FIFO order, it is not suitable to be used in time constrained systems.
2. MINIX hides interrupts using message transfers. On each hardware interrupt, a message is sent to an IST forcing a context switch before running the task. This fact increases the system latency and reduces the schedulability of RT-tasks.

7. RT-MINIXv2 System Architecture

As RT-MINIXv2 intends to be used in an academic environment, its design has been done to be as least intrusive as possible in the standard MINIX source code. Yodaiken and Barabanov [4] have proposed that approach for RTLinux. The key idea is how interrupt management is done.

As result, one RTOS (RT-MINIXv2) hosts a standard time sharing OS (MINIX). These two OS have their own set of system calls.

The RT-MINIXv2 effectively puts in place a new process scheduler that treats the MINIX kernel as the lowest priority process executing under the RT-kernel.

As Non Real-Time - (NRT) interrupt handlers could block RT-Process or RT-interrupt handlers, the RT-microkernel installs an interrupt dispatcher that only executes the handler if it has a greater priority than the running process/handler.

Under that design, MINIX only executes when there is no RT-process to run, and the RT-kernel is inactive. Thus, a MINIX process can never block interrupts or prevent itself from being preempted, yielding all resources to a RT-process. MINIX kernel may be preempted by a RT-process even during a system call, so no MINIX routine can be safely called from a RT-process.

Thus, some problems must be solved:

- 4 The interrupts must be captured by a RT-kernel.
- 4 RT-scheduler and services must be implemented.
- 4 RT-applications need an interface layer to interact with the RT-kernel.
- 4 RT-applications may need transfer data and synchronize with NRT-applications.
- 4 Full process and interrupt handler preemptability is needed.

8. RT-MINIXv2 Interrupt Handling

As RTLinux does, RT-MINIXv2 uses the Virtual Machine (VM) concept limited to interrupt emulation or virtualization. Its microkernel is underneath of MINIX and the scheduler runs NRT-processes when there are not RT-processes ready to run.

Since interrupts can come at any time, the kernel might be handling one of them while another one (of a different type) occurs. This should be allowed as much as possible since it keeps the I/O devices busy. As a result, the interrupt handlers must be coded to run in a nested manner.

When each interrupt handler terminates, the kernel must be able to resume execution of the interrupted process or switch to another process if the interrupt signal has caused a rescheduling activity or execute another lower priority interrupt handler.

Although the kernel may accept a new interrupt signal while handling a previous one, some critical regions exist inside the kernel code where interrupts must be disabled. Such critical regions must be limited as much as possible since, the kernel, and in particular the interrupt handlers, should run most of the time with the interrupts enabled.

RT-MINIXv2 avoids disabling interrupts for extended periods to improve system response time. RT-interrupt handlers can easily be replaced with standard ones. This is especially useful in certain debugging situations.

RT-MINIXv2 starts in Non Real-Time processing mode. In this mode, only NRT-interrupt handlers are executed and a limited number of RT-system calls are allowed.

To start the RT-processing mode, the *rtm_RTstart()* system call must be executed. In this mode, when an interrupt occurs, a RT-handler is invoked for RT-defined interrupts; otherwise standard MINIX handlers are called.

8.1. Interrupt Virtualization

One of the problems with doing hard Real-Time on a standard MINIX system is the fact that the kernel uses disabling interrupts as a means of synchronization. Promiscuous use of disabling and enabling interrupts inflicts unpredictable interrupt dispatch latency.

In RT-MINIXv2, this problem has been solved by adding a layer of emulation software between the MINIX kernel and the interrupt controller hardware. The emulator catches all hardware interrupts.

All MINIX kernel functions that handle the processor Interrupt Flag (IF) (*lock()/unlock()*) are replaced with virtualized ones to avoid that MINIX could not be preemptive when a RT-interrupt occurs.

All MINIX kernel functions that operate on interrupt handlers are virtualized. These functions are:

- *intr_init():*
  - Initializes the 8259 Programmable Interrupt Controller (PIC).
  - Initializes counters, indexes, etc and the table of RT-interrupt handlers.
  - Initializes the interrupt pending queues bitmap.
  - Clears the interrupt pending queues.
- *put_irq_handler():*
  - Registers an interrupt handler.
  - Inserts a NRT-handler in the interrupt pending queue.

The task of capturing and redirecting the interrupts was addressed by creating a small RT-microkernel, which captures all hardware interrupts and redirects them to either standard MINIX handlers or to RT-MINIXv2 handlers.

The RT-microkernel provides a framework onto which RT-MINIXv2 is mounted with the ability to fully preempt MINIX.

A drawback is that the MINIX kernel suffers a slight performance loss when RT-MINIXv2 VM is added due to:

- The redirections of interrupt handlers to a common interrupt dispatcher.
- The interrupt mask/unmask functions.
- The search of pending interrupts in the interrupt pending queues.
- The deferred execution of interrupt handlers.
- The added statistics collection as part of interrupt handling.

In consideration of both strengths and weaknesses, this technique has shown itself to be flexible because it removes none of the capability of standard MINIX, yet it provides guaranteed scheduling and response time for critical tasks.

Changes to standard MINIX are minimal with the VM approach. This low level of intrusion on the standard MINIX kernel improves the code maintainability to keep the Real-Time modifications up-to-date with the latest release of the MINIX kernel.

8.2. User and Kernel Stacks

In MINIX and RT-MINIXv2 each process has two stacks:

- *User stack*: In userspace, only this stack can be used.
- *Kernel stack*: When entering the kernel, the system switches to this stack.

On interrupt, the state of a process is saved in kernel stack. If new interrupts occur during the service of other interrupts (nested interrupts), the state of the interrupted handler is saved in the kernel stack.

To let monitor the kernel stack use, each interrupt descriptor have a field named *reenter* that keeps the maximum kernel reentrancy level for each IRQ. This helps to size the kernel stack for specific uses.

As is expected that a RTOS receives much more interrupts than a time-sharing OS, by default, RT-MINIXv2's kernel stack doubles in size MINIX's kernel stack.

8.3. Interrupt Handler Types

Not all interrupts have the same urgency. In fact, the interrupt handler itself is not a suitable place for all kind of actions. Long non-critical operations should be deferred, since while an interrupt handler is running, the signals on the corresponding IRQ line are ignored.

RT-MINIXv2 defines the following types of hardware interrupt handlers:

- *Non RT-handler:* The handler is executed only if its priority is greater than the priority of the interrupted process otherwise it is signaled for later processing.
- *RT Event-Driven handler:* Idem Non RT-handler.
- *RT Timer-Driven handler:* It is signaled to be for deferred processing once it reaches its period.

One special case is the Timer handler that executes some actions when the timer interrupt occurs and other are deferred as the control of timers, alarms, Timer-Driven IRQs and periodic processes.

8.4. Interrupt Service Routines

At startup, RT-MINIXv2 initializes the Interrupt Descriptor Table (IDT) pointing each entries of master PIC hardware interrupts to a routine generated by the macro *hwint_master(irq).* The entries for the slave PIC hardware interrupts are filled with the address of a routine generated by the macro *hwint_slave(irq).*

All interrupt service routines perform the same basic actions:

1. Save the registers´ contents in the Kernel Mode stack.
2. Increase the kernel variable *k_reenter* (initialized in –1).
3. If *k_reenter = 0*, the state of the User Mode process is saved, otherwise the system is already in kernel mode.
4. Send an acknowledgment to the PIC that is servicing the IRQ line, thus allowing it to issue further interrupts.
5. Execute the interrupt handler dispatcher *RTM_IRQ_dispatch().*
6. Terminate by jumping to the *restart* label if the *k_reenter = 0* or to *restart1* label for *k_reenter > 0*. More details in Section 8.9 (*Returning from System Calls and Interrupts Service Routines)*.

The *k_reenter* variable counts the reentrancy level of interrupts and system calls in the kernel.

4 *k_reenter* = –1: when the system is in User Mode.

4 *k_reenter* = 0: when one kernel control path is running. That can be a system call, an exception/fault handler or an interrupt service routine.

4 *k_reenter* > 0: when more than one kernel control path is running. This occurs on nested hardware interrupts.

8.5. Interrupt Descriptor Data Structure

This section explains the data structures that support interrupt handling and how they are laid out in various descriptors used to store information on the state, statistics and behavior of interrupt handlers.

The kernel has its own interrupt descriptor table (other than INTEL's IDT) named *RTM_desc_table[].* It is an array of *RTM_irq_desc_t* data structures. It has one descriptor for each hardware or software IRQ.

The *RTM_irq_desc_t* data structure has the following fields:

- *nrthandler*: A pointer to a function that is the NRT-handler.
- *rthandler*: A pointer to a function that is the RT-handler.
- *irq*: The IRQ number.
- *count*: An interrupt counter for statistics.
- *rain*: A counter for timer driven interrupts since the last period.
- *maxrain*: Stores the maximum value of *rain.*
- *timestamp*: The last interrupt timestamp.
- *latency*: The estimated interrupt handler latency in timer Hz.
- *maxlat*: The maximum value of *latency*.
- *before*: An auxiliary field that stores the timer-2 latch counter of the Programmable Interrupt Timer (PIT) in Hz on *RTM_IRQ_dispatch()* entry.
- *reenter*: Stores the maximum value of *RTM_prtylvl* (explained in Section 8.6)
- *period*: The processing period for a Timer Driven (TD) interrupt handler in timer ticks units.
- *task*: The ID number of the RT-IST to send a message for deferred processing.
- *priority*: The priority of the handler
- *irqtype*: The type of handler. It must be:
  - *RTM_NRTINT*: for Non Real-Time handlers.
  - *RTM_TDINT*: for Timer-Driven handler.
  - *RTM_EDINT*: for Event-Driven handlers.
  - *RTM_SOFTIRQ*: for Software IRQ handlers.
- *flags: S*ome status flags.
- *next:* A pointer to the next IRQ descriptor in the interrupt pending queue.
- *prev: A* pointer to the previous IRQ descriptor in the interrupt pending queue.
- *name:* The name of the handler.

8.6. Interrupt Handler Dispatching

RT-MINIXv2 VM sets all ISRs to call an interrupt dispatcher function named *RTM_IRQ_dispatch()* that uses the interrupt descriptor table *RTM_desc_table[]* to process RT and NRT interrupts. See Figure 2.

A kernel variable called *RTM_prtylvl* stores the current system priority level of execution. This variable is set to the current process priority or to the running interrupt handler priority. It is used to reduce the priority inversion problem deciding on the execution of an interrupt handler or to defer it.

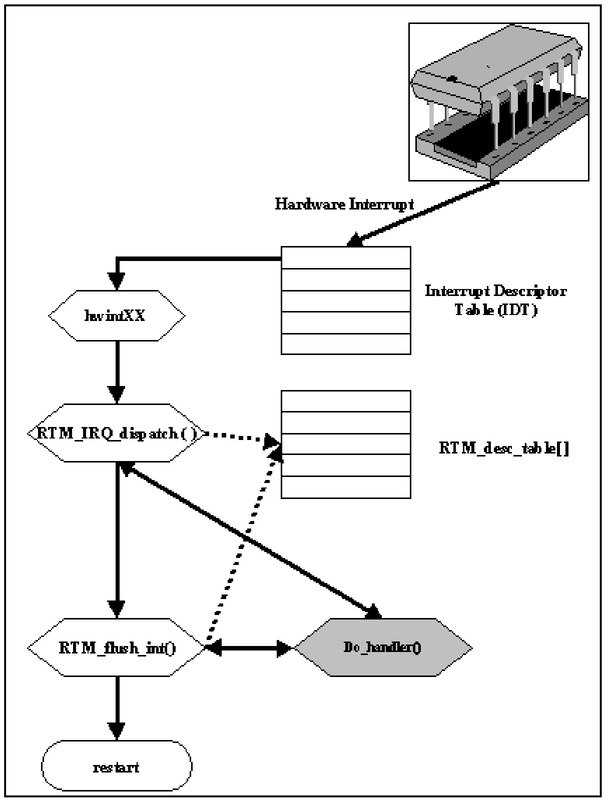

Figure 2– Interrupt Handling

If RT-MINIXv2 is running in NRT-mode, the standard interrupt handlers are executed without any interception, deferring and statistics collection.

In RT-mode, the function *RTM_IRQ_dispatch()* perform the following actions:

- If a flag for latency computation has been set in the descriptor, the timer-2 of the PIT is read before running the handler and its value is stored in the *before* descriptor field.
- Some statistics are collected as:
  - The total number of interrupts.
  - A counter for each kind of IRQ.
  - The kernel reentrancy level when the interrupt occurs.
- If a Timer interrupt has occurred:
  - The system tick counter *RTM_counter.ticks* is increased.
  - The interrupt descriptor *timestamp* field is set.
  - Timer interrupt descriptor is marked for deferred processing.
- For interrupts other than Timer, the interrupt descriptor *timestamp* field is set to *RTM_counter.ticks.*
- If an Event-Driven (ED) interrupt has occurred with its priority greater than or equal to *RTM_prtylvl*, the handler is called. The handler can set signal the kernel of the need of calling the RT-scheduler on exit.
- If an ED-interrupt has occurred with its priority lower than *RTM_prtylvl,* the descriptor is marked for deferred processing.
- If a NRT-interrupt has occurred with its priority greater or equal than *RTM_prtylvl*, the handler is called and then exits.
- If a NRT-interrupt has occurred with its priority lower than *RTM_prtylvl*, its descriptor is marked for deferred processing.
- If a TD-interrupt has occurred, a bit in kernel bitmap variable named *RTM_TD_bitmap* is set to signal the Timer interrupt handler that the TD-interrupt period must be checked. Later, when the Timer Interrupt handler runs, the TD-descriptor is signaled for deferred processing once it reaches its period.

8.7. Interrupt Handler Priority

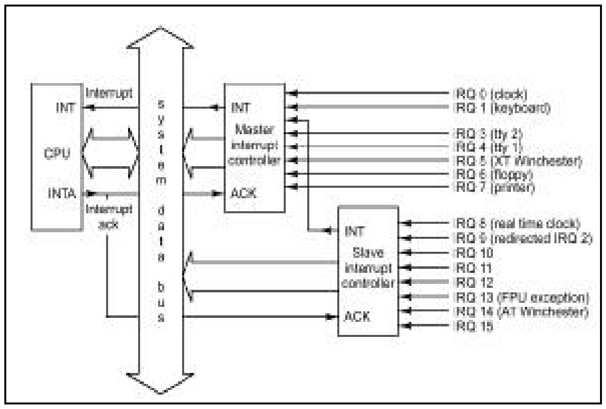

Figure 3– IRQ priorities (from [1])

The PIC treats interrupts according to their priority level. This depends on the interrupt line they use to enter the interrupt controller. For this reason, the priority levels are directly tied to the interrupt number. Higher-priority IRQs may improve the performance of the devices that use them.

The standard PC hardware has assigned priorities for standard interrupts related to IRQ number as shown in Figure 3. A lower IRQ number implies higher priority. Newer Advanced Programmable Interrupt Controllers (APICs) allow programmers to change the priority of each IRQ line. RT-MINIXv2 services each IRQ based on the *priority* field of a descriptor.

If a lower priority interrupt occurs during the execution of a running process or handler, its interrupt descriptor is marked as triggered and its processing is delayed. Later, after a context switch, system call or return from interrupt, the kernel calls *RTM_flush_int()* function to scan the queues for interrupt descriptors that has been triggered to invoke their handlers.

RTM_flush_int() does not execute some pending interrupt handlers . They are:

- NRT-interrupts once MINIX has (virtually) disabled processor interrupts using the *lock()* function.
- TD-interrupts that have not reach their launching period.
- All pending interrupts handlers with lower priority than the current system priority level of execution (*RTM_prtylvl*).

Each RT-handler must signal if it needs RT-scheduler invocation on exit.

8.8. Software IRQs

One of the main problems with interrupt handling is how to perform longish tasks within a handler.

Often a substantial amount of work must be done in response to a device interrupt, but interrupt handlers need to finish up quickly and not keep interrupts blocked for long. These two needs (work and speed) conflict with each other, leaving the driver writer in a bit of a bind. Therefore it is desirable that the interrupt handlers could delay the execution of some tasks so that they do not block the system for too long.

As it is explained above, MINIX uses a two stages interrupt management where an interrupt handler partially process the interrupt and then sends a message to a device driver task to resume the interrupt processing. This approach implies at least a context switch to restore the state of the IST.

Linux kernel resolves this problem by splitting the interrupt handler into two halves. The so-called top half is the routine that actually responds to the interrupt. The bottom half is a routine that is scheduled by the top half to be executed later, at a safer time.

In RT-MINIXv2, device driver writers could decide among three approaches:

- Complete interrupt processing in the handler.
- Two stages interrupt management like MINIX using ISTs and message transfers.
- Two stages interrupt management using Software IRQs as Linux does.

Triggered Software IRQs are kernel routines that are invoked by *RTM_flush_int()* as it do with pending hardware interrupts.

Software IRQ descriptors have the same data structure *RTM_irq_desc_t* than Hardware IRQ descriptors; therefore they have priority, counters, timestamps and other fields.

The motivation for introducing software IRQs is to allow a limited number of functions related to interrupt handling to be executed in a deferred manner. This increases the system responsiveness because some work is executed out of interrupt time. Additionally the processing overhead is lower than using the IST model because it avoids the context switch among the interrupted process and the IST.

Software IRQ can be used as kernel threads triggered by time using *virtual clocks* (system virtual timers).

8.9. Returning from System Calls and Interrupts Service Routines

Although the main objective of the termination phase of interrupt and exception handlers is to resume execution of some program, several issues must be considered before doing it:

- The number of kernel control paths being concurrently executed: if there is just one (*k_reenter* = 0), the CPU must switch back to User Mode.
- If there are Triggered Interrupt descriptors that were deferred, their handlers are executed. This is accomplished by the *RTM_flush_int()* kernel function.
- If (k_reenter = 0), the kernel must flush any held-up interrupt messages from handlers to ISTs.

8.10. Flushing Deferred Interrupts

Before returning the CPU to User Mode those interrupt handlers that has been deferred must be executed. The kernel function that accomplishes with these issues is *RTM_flush_int().*

RT-MINIXv2 uses a system of priority ordered interrupt queues based on a bitmap named *RTM_irqQ_bitmap*. A bit set in the i-th position of the bitmap implies that at least one descriptor in the i-th interrupt pending queue has been triggered for deferred processing as it can see in Figure 4.

RTM_flush_int() scans the queues for hardware and software IRQ handlers to run. The scan stops when the priority of the queue is lower than *RTM_prtylvl*.

A handler is executed only if its descriptor status flags have the *RTM_TRIGGERED* bit set.

Once all triggered interrupt descriptors in the same queue has been serviced, the bit in the interrupt queue bitmap is cleared and if one of the handlers has set that it need rescheduling, and the RT-scheduler is invoked.

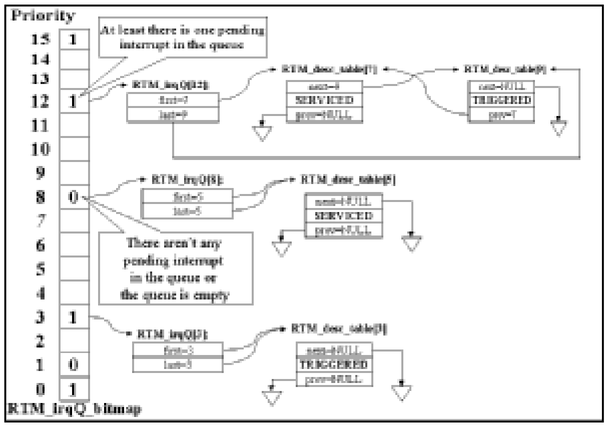

Figure 4 - IRQ bitmap and IRQ Priority Queues

As during the execution *RTM_flush_int()*, new interrupts can occur, a lock bit in the kernel variable *RTM_flags* is set to avoid that *RTM_flush_int()* could be called again at *RTM_IRQ_dispatch()* exit. If the new interrupt have greater priority than the interrupted handler, another bit in *RTM_flags* is set to signal *RTM_flush_int()* that it needs to restart the scan again from the highest priority interrupt queue to find the new higher priority triggered interrupt descriptor.

8.11. Running the Handler

Before running the handler:

- 4 The kernel saves the current system priority level of execution *RTM_prtylvl*.
- 4 The *RTM_prtylvl* is set to the handler priority.
- 4 The CPU interrupts are enabled.

Once the handler exits:

- 4 The CPU interrupts are disabled.
- 4 An estimation of the handler latency could be computed (see Section 8.14).

For hardware interrupts, the value returned by the handler signal the kernel if the IRQ line in the PIC must be enabled.

8.12. Interrupt Descriptor Timestamp

RT-MINIXv2 includes a timestamp field in the interrupt descriptor data structure *RTM_irq_desc_t* that is set by *RTM_IRQ_dispatch()* to *RTM_counter.ticks*.

The units of the timestamp field are RT-ticks since the execution of the *rtm_rtstart()* or *rtm_restart()* system call execution.

The Timer interrupt handler uses the descriptor timestamp field to decide about triggering Timer-Driven interrupt handlers.

8.13. Kernel Functions for Interrupt Handling

RT-MINIXv2 microkernel has the following functions to handle interrupts:

- 4 *RTM_lock()*: it is the true CLI assembler instruction to disable interrupts at the CPU.
- 4 *RTM_unlock()*: it is the true STI assembler instruction to enable interrupts at the CPU.
- 4 *RTM_set_irqd()*: sets all fields of an interrupt descriptor. This function inserts the descriptor in one of the interrupt pending queues too.
- 4 *RTM_free_irqd()*: removes an interrupt descriptor from its interrupt pending queue. It resets all fields of an interrupt descriptor. The RT-handler and NRT-handler points to the *spurious_irq()* handler.
- 4 *RTM_set_softirq()*: assigns a new interrupt descriptor from the software IRQ descriptor pool, then sets the descriptor using *RTM_set_irq()*.
- 4 *RTM_free_softirq():* removes the descriptor from the interrupt pending queue using *RTM_free_irqd()* and free the descriptor to the software IRQ descriptor pool.

8.14. Estimating Handlers Latency

One of the most common measurements requested of RT-kernel and RTOS is the interrupt processing time latency [11]. This metric, while useful, is a very limited indication of the performance capabilities of a RT-kernel. RT-environments require the ability to predict how fast an OS will react to interrupts.

Interrupt processing time latency numbers are most useful when used to measure the effectiveness of a RT-kernel at dealing with extremely high-priority interrupts or emergency interrupts. Often a peripheral must be serviced within a certain time limit after an event. For example, a packet must be read from a network port before the next on arrives.

The interrupt processing time latency is the maximum time taken to respond to an interrupt request. This will include the time it takes for the current instruction to complete and the time for the CPU to respond to the interrupt. If an ISR is already executing and cannot be interrupted then this also increases the interrupt processing time latency.

Understanding the relative size of delays is important to the design of the RT-system. Most sources of delay in a RT-kernel are due to either code execution or context switches. Virtually all of these delays are fixed in length and repeatable. Bounded and repeatable is the fundamental characteristic desired of a RT-kernel.

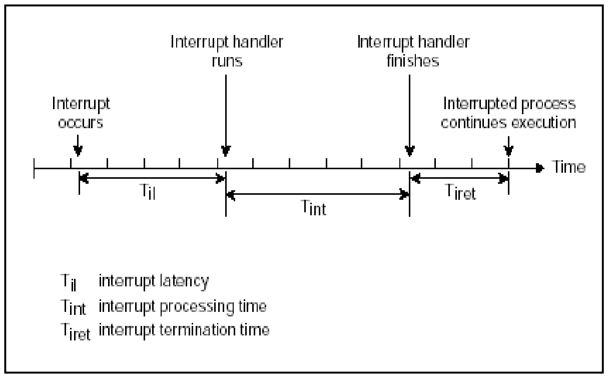
Interrupt handler runs
Interrupt handler finishes
Interrupt occurs
Interrupted process continues execution
Time
T_{il}
T_{int}
T_{iret}
T_{il} interrupt latency
T_{int} interrupt processing time
T_{iret} interrupt termination time

Figure 5 - Interrupt Latency (from [12])

Interrupt processing time latencies are not fixed in length. Because an interrupt is, by definition, an asynchronous event, a system's interrupt latency is dependent on the state of the machine when the interrupt occurred. This state is a function of both the hardware and the software used in the system.

RT-MINIXv2 uses the timer-2 of the Programmable Interrupt Timer (PIT) to estimates the handler latency. The PIT is programmed in SQUARE_WAVE mode and the divisor latch is set to 65536, therefore its period is 65536/1193182 = 0.0549 seconds.

At the start of execution of *RTM_IRQ_dispatch()*, the value of the counter of the timer-2 latch of the PIT is stored in the *before* field of the interrupt descriptor. Once the handler finishes its execution, the counter is read again and the latency of the handler could be estimated as the difference between these two values. The unit of the estimated latency is PIT Hz or $(1193182)^{-1}$ or 0.838 [µs].

The estimated latency by this method represents the Interrupt processing time (Tint) in Figure 5.

As PIT timer-2, the timer-0 is programmed in SQUARE_WAVE mode. The PIT timer-0 signals the Timer Interrupt once the counter reaches zero. As the PIT remains decreasing the counters, during the execution of the Timer handler, the value of the timer-0 counter lets compute the latency of the handler.

8.15. Nested Interrupts

As has been mentioned above, RT-MINIXv2 support nested interrupts. Worst-case timing considerations for unmasked interrupts must be included in the computation of the service time for all interrupts currently being processed.

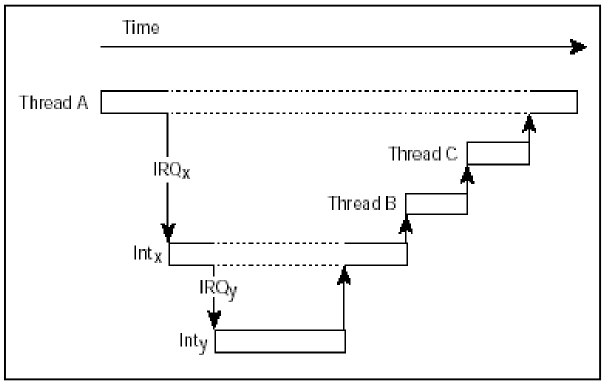

Figure 6- Nested Interrupts (from [12])

The following criteria are established for nested interrupts:

4 A higher (or equal) priority IRQ will preempt a running handler. The computation time of the higher priority handler must be added to the service time as a blocking time.

4 A lower priority IRQ will not preempt a running handler. The execution of the new IRQ handler will be deferred for later processing and the function *RTM_flush_int()* will not be invoked on exit to reduce the blocking time. This blocking time is a priority inversion that must be added to the running handler service time. It must be consider only once because the lower priority IRQ will be disable at PIC level until the handler will re-enable it later.

Once the handler exits, the RT-scheduler could be invoked and another process/thread preempts the former as threads B and C that preempt thread A in Figure 6.

At present, the author is evaluating the performance impact on including an interrupt mask for process/ handlers descriptors that lets change the PIC mask before running a handler/process to avoid that could be interrupted by a lower priority IRQ.

8.16. Timer-Driven Interrupts

This kind of interrupt processing lets that more than one interrupt (a rain) could occur in a time period without running the handler. The handler only is executed at specified periods reducing system overload.

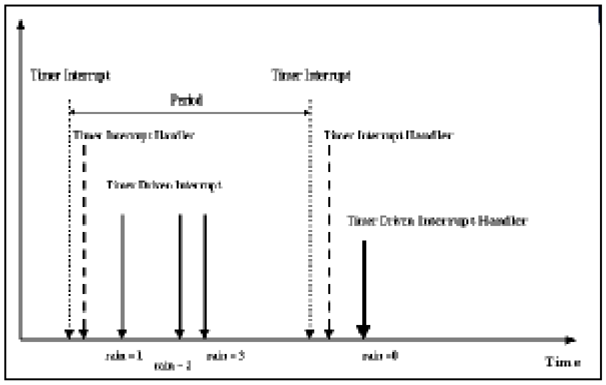

Figure 7- Timer Driven Interrupts

The processing of a TD-interrupt handler has 3 stages (see Figure 7):

1. The TD-interrupt occurs and a bit is set in the *RTM_TD_bitmap* to signal the Timer Interrupt to trigger the TD-descriptor when it reaches its period. The descriptor field *rain* that counts the number of interrupts since the last period has been increased.
2. The Timer interrupt occurs and it checks all bits in *RTM_TD_bitmap* for TD-interrupts since the last period. If the period of a TD-descriptor has been met, it is triggered to be processed by *RTM_flush_int()* same as other delayed interrupt handlers.
3. *RTM_flush_int( )* runs the handler and resets the *rain* field of the descriptor.

9. Conclusions

MINIX proved to be a feasible testbed for OS development and RT-extensions that could be easily added to it.

RT-MINIXv2 Virtual Machine architecture and source code readability make it suitable for course assignments and Real-Time project developments. Its microkernel offers device-drivers writers Event-Driven, Timer-Driven, Software IRQs and Non-Real-Time interrupt handler types. These features make it a good choice to conduct RT-experiences.

10. Future Works

The author is working on to the following topics:

4 *Process Management*: to support periodic and sporadic RT-processes.

4 *Time Management*: This topic covers the design and implementation of virtual timers to be used for timeouts, alarms, periodic execution, etc.

4 *Process Scheduling*: It covers a new process scheduler with 16 priority levels.

4 *Interprocess Communications*: A new set of RT-primitives to communicate RT-processes supporting the Priority Inheritance Protocol.

4 *Statistics Collection*: As RT-MINIXv2 is intended to be used in academic environments, statistics collections is an important issue.

Other planed projects based on RT-MINIXv2 are:

4 */rtproc Filesystem*: equivalent to Linux's */proc* Filesystem.

4 RT-FIFOs: equivalents to RTLinux RT-FIFOs that communicates RT-proccess with NRT-process.

4 *IRQ sharing*: PCI requires shared interrupts, therefore this features is intended to support multiple interrupt handlers for the same IRQ.

11. Acknowlegements

The author gratefully acknowledges help received from:

4 Telecom Argentina S.A. for sponsoring the author's UNLP Master.

4 German Maglione for supplying the IBM Thinkpad 370 notebook used in the implementation of RT-MINIXv2.

4 Nicolas Cesar and Felipe Ieder for the implementation of system latency tests.

4 Dr. Silvio Gonnet for the revision of this article.

12. References

[1] Tanenbaum Andrew S., Woodhull Albert S., *Sistemas Operativos: Diseño e Implementación 2da Edición*, ISBN 9701701658, Editorial Prentice-Hall , 1999

[2] Pablo J. Rogina - Gabriel Wainer., “New Real-Time Extensions to the MINIX operating system”, *Proc. of 5 th Int. Conference on Information Systems Analysis and Synthesis (ISAS'99)*,August, 1999.

[3] Gabriel A. Wainer, “Implementing Real-Time services in MINIX”, *ACM Operating Systems Review*, July 1995.

[4] Victor Yodaiken, Michael Barabanov, “A Real-Time Linux”, ”, *Proceedings of Linux Applications Development and Deployment Conference (USELINUX)*, January, 1997 available online at *http://rtlinux.cs.nmt.edu/*

[5] Intel Corporation, *Intel Architecture Software Developer's Manual Volume 3: System Programming Guide,* 1997

[6] Victor Yodaiken, “Cheap Operating System Research and Teaching with Linux:”, available online at http://citeseer.ist.psu.edu/75556.html

[7] Paul Ashton, Carl Cerecke,Craig McGeachie, Stuart Yeates, “Use of interaction networks in teaching Minix” *Technical Remailbox . COSC 08/95*, Dept. of Computer Science . University of Canterbury,1995.

[8] Paul Ashton, Daniel Ayers, Peter Smith.SunOS “MINIX: A tool for use in Operating System laboratories”, *Technical Remailbox, Australian Computer Science Communications*, 16(1): 259-269, 1994.

[9] Stephen J Hartley, “More Experience with MINIX in Operating System lab”, available online at ftp://ftp.mcs.drexel.edu/pub/shartley/minix.PO.gz.

[10] Daniel P. Bovet, Marco Cesati, *Understanding the Linux Kernel Second Edition,* 2003 - O'Reilly – 2003.

[11] Radisys, “INtime Interrupt Latency Report”, available on line at http://www.profimatics.de/products/intime/manuals/intime.interrupt.latency.report.pdf

[12] QNX Software Systems Ltd. 2002 – “QNX Neutrino Realtime Operating System – System Architecture”, available on line at http://www.mikecramer.com/Qnx/momentics_nc_docs/neutrino/sys_arch/kernel.html

[13] RTnet- Hard Real-Time Networking for Linux/RTAI, available at http://www.rts.uni-hannover.de/rtnet/index.html

[14] Rether: A Real-Time Ethernet Protocol, available at http://www.ecsl.cs.sunysb.edu/rether/